AD-A034 982 ILLINOIS UNIV AT URBANA-CHAMPAIGN COORDINATED SCIENCE LAB F/G 12/1
A SEMI-FAST FOURIER TRANSFORM ALGORITHM OVER GF (2 TO THE MTH P--ETC(U)
SEP 76 D V SARWATE DAAB07-72-C-0259
UNCLASSIFIED R-735 NL

1 OF 1
ADA034982

END
DATE
FILMED
3 - 77

(12)

b.s.

REPORT R-735 SEPTEMBER, 1976 UILU-ENG 76-2223

CSL COORDINATED SCIENCE LABORATORY

A SEMI-FAST FOURIER TRANSFORM ALGORITHM OVER GF(2^m)

DILIP V. SARWATE

UNIVERSITY OF ILLINOIS – URBANA, ILLINOIS

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE *(When Data Entered)*

REPORT DOCUMENTATION PAGE		READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER	2. GOVT ACCESSION NO.	3. RECIPIENT'S CATALOG NUMBER
4. TITLE *(and Subtitle)* (6) A SEMI-FAST FOURIER TRANSFORM ALGORITHM OVER GF(2^m). 2 to the mth power		5. TYPE OF REPORT & PERIOD COVERED (9) Technical Report
		6. PERFORMING ORG. REPORT NUMBER (14) R-735, UILU-ENG-76-2223
7. AUTHOR(s) (10) Dilip V. Sarwate		8. CONTRACT OR GRANT NUMBER(s) (15) DAAB 07-72-C-0259
9. PERFORMING ORGANIZATION NAME AND ADDRESS Coordinated Science Laboratory University of Illinois at Urbana-Champaign Urbana, Illinois 61801		10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS
11. CONTROLLING OFFICE NAME AND ADDRESS Joint Services Electronics Program		12. REPORT DATE (11) September, 1976
		13. NUMBER OF PAGES 21
14. MONITORING AGENCY NAME & ADDRESS *(if different from Controlling Office)* (12) 26p.		15. SECURITY CLASS. *(of this report)* UNCLASSIFIED
		15a. DECLASSIFICATION/DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT *(of this Report)*

Approved for public release; distribution unlimited

17. DISTRIBUTION STATEMENT *(of the abstract entered in Block 20, if different from Report)*

18. SUPPLEMENTARY NOTES

19. KEY WORDS *(Continue on reverse side if necessary and identify by block number)*

Fast Fourier Transforms
finite fields
computational complexity
analysis of algorithms
error-correcting codes

(2 to the mth power)

n squared

20. ABSTRACT *(Continue on reverse side if necessary and identify by block number)*

An algorithm which computes the Fourier Transform of a sequence of length n over GF(2^m) using approximately 2nm multiplications and n^2 + nm additions is developed. The number of multiplications is thus considerably smaller than the n^2 multiplcations required for a direct evaluation, though the number of additions is somewhat larger. Unlike the Fast Fourier Transform, this method does not depend on the factors of n and can be used when n is not highly composite or is a prime. The bit complexity of the algorithm is analyzed in detail. Implementations and applications are briefly discussed.

n squared

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE

UNCLASSIFIED 097 700

SECURITY CLASSIFICATION OF THIS PAGE *(When Data Entered)*

LB

Index terms: Fast Fourier Transforms, finite fields, computational complexity, analysis of algorithms, error-correcting codes

UILU-ENG 76-2223

A Semi-Fast Fourier Transform Algorithm Over GF(2^m)

by

Dilip V. Sarwate

This work was supported in part by the Joint Services Electronics Program (U.S. Army, U.S. Navy and U.S. Air Force) under Contract DAAB-07-72-C-0259.

A Semi-Fast Fourier Transform Algorithm over GF(2^m)

by

Dilip V. Sarwate*

July 28, 1976

Abstract

An algorithm which computes the Fourier Transform of a sequence of length n over GF(2^m) using approximately 2nm multiplications and n^2 + nm additions is developed. The number of multiplications is thus considerably smaller than the n^2 multiplications required for a direct evaluation, though the number of additions is somewhat larger. Unlike the Fast Fourier Transform, this method does not depend on the factors of n and can be used when n is not highly composite or is a prime. The bit complexity of the algorithm is analyzed in detail. Implementations and applications are briefly discussed.

*The author is with the Coordinated Science Laboratory and the Department of Electrical Engineering, University of Illinois at Urbana-Champaign, Urbana, Illinois 61801.

This work was supported in part by the Joint Services Electronics Program (U.S. Army, U.S. Navy and U.S. Air Force) under Contract DAAB-07-72-C-0259.

I. Introduction

The Fast Fourier Transform (FFT) algorithm of length n over a finite field [1] is essentially the well-known complex field FFT algorithm (e.g. [2]) with the primitive n^{th} root of unity $\exp(j2\pi/n)$ in the complex field being replaced by a primitive n^{th} root of unity in the finite field (or an extension thereof). When n is composite, with factors $n_1, n_2, \ldots, n_s$, the finite field FFT is essentially what is called a mixed-radix FFT and requires $n(n_1 + n_2 + \ldots + n_s)$ multiplications and $n(n_1 + n_2 + \ldots + n_s)$ additions as compared to the n^2 multiplications and n^2 additions required to evaluate the Fourier Transform in the most obvious way. If n is not highly composite, (or if n is a prime), the saving in computation is quite small (or nonexistent). In such cases, the Fourier Transform can be computed from the cyclic convolution of two appropriately defined sequences of length approximately 2n or more [3] - [5]. This convolution itself can be computed by computing the forward transforms of the two sequences, a pointwise multiplication of the transforms, and an inverse transform. If the length of the sequences is chosen to be highly composite, the FFT algorithm can be used to compute the three transforms and significant savings in computation can be achieved.

For small to moderate values of n, however, the cyclic convolution technique can be slower than direct evaluation. One other problem that arises with finite fields is that the finite field may not contain an appropriate primitive root of unity, and computations may have to be done in a much larger field. For example, if we wish to compute the transform of a sequence of 31 elements from $GF(2^5)$, we can find it from the cyclic convolution of two sequences of length (say) 63. Unfortunately, the smallest field that contains $GF(2^5)$ as well as a primitive 63rd root of unity is $GF(2^{30})$ [6], [7]. Of course, one might cyclically convolve two sequences of length 93, which will allow the use of $GF(2^{10})$. On the other hand, 93 is not highly composite. In fact, computing three FFTs of length 93 (or 63) requires more computation in a larger field than a brute-force evaluation of the original transform. The situation is somewhat better if one redefines the problem so that computation can be done in $GF(p)$ for some large prime p or in the complex field itself. However, these techniques will not be analyzed in this paper.

The algorithm proposed in this paper requires $2(n-1)(m-1)$ multiplications and $(n-1)(n+m-1)$ additions in $GF(2^m)$ to compute a transform of length n, n a prime, over $GF(2^m)$. If n is not a prime, the number of multiplications is somewhat less and the number of additions is somewhat more. Since multiplications require more time than additions, the algorithm is somewhat faster than the direct method, though both have arithmetic complexity of the same order $O_A(n^2)$. The bit complexity of the proposed algorithm is $O_B(n^2 \log n)$ which is better by a factor of log n over the direct method. For small values of n, the proposed method is also superior to the cyclic convolution technique. Asymptotically, of course, the cyclic convolution technique using the FFT has arithmetic complexity $O_A(n \log n)$ and bit complexity $O_B(n \log^5 n)$, and is vastly superior. For these reasons, the proposed algorithm is dubbed a Semi-Fast Fourier Transform (S-FFT) algorithm.

II. The S-FFT Algorithm

Let n be an odd integer,

m the multiplicative order of 2 modulo n

α a primitive n^{th} root of unity in $GF(2^m)$

β an element of degree m in $GF(2^m)$. It is convenient, but not necessary, to take β to be a primitive element. In fact, one can take $\beta = \alpha$.

Let $A(x) = \sum_{i=0}^{n-1} A_i x^i$ be a polynomial over $GF(2^m)$.

Every element of $GF(2^m)$ can be represented as a polynomial of degree less than m in β (see, e.g. [7]). Thus,

$$A_i = \sum_{k=0}^{m-1} A_{i,k} \beta^k \qquad \text{where } A_{i,k} \in GF(2) \tag{1}$$

Hence,

$$A(x) = A^{(0)}(x) + \beta A^{(1)}(x) + \beta^2 A^{(2)}(x) + \ldots + \beta^{m-1} A^{(m-1)}(x) \tag{2}$$

where $A^{(k)}(x) = \sum_{i=0}^{n-1} A_{i,k} x^i$ is a polynomial over $GF(2)$.

The Fourier Transform of $A(x)$ is $B(z) = \sum_{j=0}^{n-1} B_j z^j$ where

$$B_j = A(\alpha^j) = \sum_{i=0}^{n-1} A_i (\alpha^j)^i \tag{3}$$

Using (1), it is easy to manipulate (3) to give

$$B_j = \sum_{k=0}^{m-1} [A^{(k)} (\alpha^j)] \beta^k \tag{4}$$

If $B^{(k)}(z)$ denotes the Fourier Transform of $A^{(k)}(x)$, then, from (2) and (4), we get

$$B(z) = B^{(0)}(z) + \beta B^{(1)}(z) + \beta^2 B^{(2)}(z) + \ldots + \beta^{m-1} B^{(m-1)}(z) \tag{5}$$

The basic idea behind the algorithm is to find the polynomials $B^{(k)}(z)$ as efficiently as possible and then use (5) to compute $B(z)$. It is easier to compute B_0 directly from (3) using $(n-1)$ additions rather than from (5). The rest of the coefficients of $B(z)$ can be computed from the coefficients of the $B^{(k)}(z)$'s in $(n-1)(m-1)$ multiplications and $(n-1)(m-1)$ additions using (5). We now show that the coefficients of the $B^{(k)}(z)$'s can be computed quite rapidly because the $B^{(k)}(z)$'s are transforms of binary polynomials.

$$\begin{aligned}[B_j^{(k)}]^2 &= \left[\sum_{i=0}^{n-1} A_{i,k}(\alpha^j)^i\right]^2 \\ &= \sum_{i=0}^{n-1} A_{i,k}^2(\alpha^j)^{2i} \qquad \text{in a field of characteristic 2.} \\ &= \sum_{i=0}^{n-1} A_{i,k}(\alpha^{2j})^i \qquad \text{since } A_{i,k} \in GF(2)\end{aligned}$$

Thus,

$$[B_j^{(k)}]^2 = B_{2j}^{(k)} \tag{6}$$

Given $B_j^{(k)}$, one can compute $B_{2j}^{(k)}$, $B_{4j}^{(k)}$, $B_{8j}^{(k)}$, ..., etc. (subscripts taken modulo n) simply by successive squarings. Note that α^j, α^{2j}, α^{4j}, ..., etc.

are conjugate elements in $GF(2^m)$ and are the roots of the same irreducible polynomial. Let $I(n)$ denote the number of such irreducible polynomials (of degree greater than 1) that are divisors of x^n-1. If we compute $B_j^{(k)}$ for $I(n)$ values of j, then all the other $B_j^{(k)}$'s (except $B_0^{(k)}$ which need not be computed at all) can be obtained by squaring. Now, to compute $B_j^{(k)}$ requires at most $(n-1)$ additions because the $A_{i,k}$ are 0 or 1 and, hence, either $(\alpha^j)^i$ is added to the sum or it is not. Thus, we can compute all the coefficients of $B^{(k)}(z)$ except $B_0^{(k)}$ in $(n-1)I(n)$ additions and $(n-1) - I(n)$ squaring operations (i.e. multiplications) in $GF(2^m)$. There are m such polynomials $B^{(k)}(z)$ and thus, we find that all the coefficients of $B(z)$ can be computed using a total of $m(n-1)(I(n)+1)$ additions and $m(n-1-I(n)) + (n-1)(m-1)$ *multiplications over* $GF(2^m)$.

Let $\phi(\cdot)$ denote Euler's ϕ function and let $\Psi(d)$ denote the multiplicative order of 2 mod d. Thus, $\Psi(d)$ divides $\Psi(n) = m$ if d divides n. Then we have [8]

$$I(n) = \sum_{\substack{d>1,\\ d|n}} \frac{\phi(d)}{\Psi(d)} \tag{7}$$

If n is a prime, then we get $I(n) = (n-1)/m$. In this case, $B(z)$ can be computed using $2(n-1)(m-1)$ multiplications and $(n-1)(n+m-1)$ additions. If n is not a prime, then, using the fact that $\sum_{d|n} \phi(d) = n$, we get that $I(n) > (n-1)/m$ and thus the number of multiplications required is somewhat less than $2(n-1)(m-1)$ while the number of additions is increased. If $n = 2^m-1$, (7) can also be written [8] as

$$I(n) = \frac{1}{m}\left[\sum_{d|m} \phi(d)2^{m/d}\right] - 2 \geq (2^m-2)/m$$

Summarizing, the algorithm works as follows. Let the I(n) irreducible polynomial divisors of $x^n - 1$ be ordered in some manner; for $\ell = 1,2,\ldots,I(n)$, let $P(\ell)$ be an integer such that $\alpha^{P(\ell)}$ is a root of the ℓth irreducible polynomial; and let $Q(\ell)$ denote the number of roots of the ℓth irreducible polynomial.

<u>Algorithm S-FFT</u>

<u>Input</u>: Polynomial $A(x) = A_0 + A_1x + \ldots + A_{n-1}x^{n-1}$ over $GF(2^m)$

<u>Output</u>: Fourier Transform $B(z) = B_0+B_1z+\ldots+B_{n-1}z^{n-1}$ of $A(x)$ over $GF(2^m)$

```
begin
    $B_0 \leftarrow \sum_{i=0}^{n-1} A_i$ ;
    for $k \leftarrow 0$ step 1 until $(m-1)$ do
        for $\ell \leftarrow 1$ step 1 until $I(n)$ do
            begin
            $B^{(k)}_{P(\ell)} \leftarrow \sum_{i=0}^{n-1} A_{i,k} \, (\alpha^{P(\ell)})^i$ ;
            for $j \leftarrow 1$ step 1 until $(Q(\ell)-1)$ do
                $B^{(k)}_{2^j P(\ell) \bmod n} \leftarrow \left[B^{(k)}_{2^{j-1} P(\ell) \bmod n}\right]^2$
            end;
    for $i \leftarrow 1$ step 1 until $(n-1)$ do
        $B_i \leftarrow \sum_{k=0}^{m-1} B_i^{(k)} \beta^k$
end
```

III. Analysis of Bit Complexity

We now analyze the S-FFT algorithm, the FFT algorithm and direct computation of the Fourier Transform to determine the numbers of bit operations required for each. We consider an implementation consisting of a large combinational network with nm inputs representing the A_i and nm outputs representing the B_j; and count the number of gates required to build such a network. For simplicity, we consider the case $n = 2^m-1$ only and also take $\alpha = \beta$.

For direct computation of the Fourier Transform, the B_j's, $j = 0,1,\ldots,n-1$ can be computed by Horner's rule as

$$B_j = ((\ldots ((A_{n-1}\alpha^j + A_{n-2})\alpha^j + A_{n-3})\alpha^j + \ldots + A_1)\alpha^j + A_0$$

For $j = 0,1,\ldots,n-1$, we need $(n-1)$ multipliers capable of multiplying their inputs by α^j. These multipliers can be implemented as linear transformations of the m-bit vector inputs (e.g. [7] p. 45). The rows of the $m \times m$ matrix of such a transformation are the m-bit representations (as in (1)) of the elements $\alpha^{j+m-1}, \alpha^{j+m-2}, \ldots, \alpha^{j+1}, \alpha^j$. If there are t ones in the matrix, the linear transformation can be implemented with $t-m$ exclusive-OR (XOR) gates. Note that multiplication by 1 corresponds to the identity transformation and thus requires no XOR gates. Consider the n $m \times m$ matrices corresponding to multiplication by $1, \alpha, \alpha^2, \ldots, \alpha^{n-1}$ which are exactly the n nonzero binary m-tuples. Each nonzero binary m-tuple occurs in m different matrices. Hence, the n matrices contain a total of $m^2 2^{m-1}$ ones and $(2^m-2)(m^2 2^{m-1}-m(2^m-1))$ XOR gates are required to implement all the multiplications.

Since $(2^m-1)(2^m-2)$ additions, each requiring m XOR gates to implement, are also required, we see that a total of $m^2 2^m(2^{m-1}-1)$ XOR gates is required to compute the Fourier Transform by the direct method. The total delay through the network is $(2^m-2)(\lceil \log_2 m \rceil + 1)$ gate delay .

Another possibility in the direct method is to compute B_j by multiplying A_i by α^{ji} simultaneously for $i = 0,1,\ldots,n-1$ and summing the products. When n is a prime, the hardware requirements are exactly the same as before. When n is not a prime, the hardware requirements actually decrease very slightly. This is because whenever j and n are not relatively prime, the set of elements $1, \alpha^j, \alpha^{2j}, \ldots, \alpha^{(n-1)j}$ used in computing B_j is not the same as the set of elements $1, \alpha, \alpha^2, \ldots, \alpha^{n-1}$. In fact, if α^j is a primitive d th root of unity (where d is some divisor of n), then each of the elements $1, \alpha^j, \alpha^{2j}, \ldots, \alpha^{(d-1)j}$ occurs exactly n/d times in the former set. As a consequence, the hardware required to compute B_j depends on the order of the element α^j. In column 4 of Table 1, we give the total hardware required to compute B_j where α^j is a primitive d th root of unity for nontrivial divisors of n. Since there are $\phi(d)$ primitive d th roots of unity, the total hardware is easily inferred from these numbers. The delay through the network is $m + \lceil \log_2 m \rceil$ gate delays which is considerably smaller than that required for Horner's rule: the saving in hardware when n is not a prime is less than 1%.

Turning to the FFT algorithm, let p be the smallest nontrivial divisor of n and let $n = pq$. Let $\underline{A} = [A_0,A_1,A_2,\ldots,A_{n-1}]$ and $\underline{B} = [B_0,B_1,\ldots,B_{n-1}]$ and define $\underline{M}_{n,\alpha}$ to be a $n \times n$ matrix with entries $a_{ij} = \alpha^{ij}, i,j = 0,1,\ldots,n-1$.

(1) n	(2) d	(3) $\phi(d)$	(4) Number of bit Operations to Compute B_j	(5) Number of Bit Operations to Compute B_j after Folding	(6) Number of Bit Operations to Compute $B_j^{(k)}$	(7) Number of Bit Operations to Compute $B_j^{(k)}$ after Folding
15	3	2	126	22	26	2
	5	4	122	38	26	6
63	3	2	1,002	42	204	4
	7	6	1,182	126	174	14
	9	6	1,156	160	190	22
	21	12	1,146	378	186	58
255	3	2	8,322	90	1,012	4
	5	4	6,418	118	706	6
	15	8	7,744	448	944	48
	17	16	8,062	530	892	52
	51	32	8,192	1,632	1,032	200
	85	64	8,200	2,728	1,024	336
511	7	6	18,679	247	1,743	15
	73	72	20,767	2,959	2,315	323
1024	3	2	45,002	122	4,082	2
	11	10	49,466	522	4,826	42
	31	30	49,622	1,494	4,742	134
	33	20	52,938	1,698	4,950	150
	93	60	51,206	4,646	5,094	454
	341	300	51,206	17,062	5,126	1,702
2047	23	22	122,453	1,365	11,381	117
	89	88	123,637	5,365	10,845	461

Table 1: Numbers of bit operations to compute B_j and $B_j^{(k)}$ where α^j is a primitive d^{th} root of unity.

computing the Fourier Transform is equivalent to computing $\underline{B} = \underline{A}\ \underline{M}_{n,\alpha}$. If we rearrange the columns of $M_{n,\alpha}$ in the order $0,p,2p,\ldots,(q-1)p,1,1+p,1+2p,\ldots,(p-1),(p-1)+p,\ldots(p-1)+(q-1)p$ to get the new matrix $\underline{M}^*_{n,\alpha}$, then $\underline{A}\ \underline{M}^*_{n,\alpha}$ is simply $\underline{B}$ with its elements permuted into the same order. However, $\underline{M}^*_{n,\alpha}$ factors into two matrices $\underline{X}$ and $\underline{Y}$ where $\underline{X}$ is a $p \times p$ array of $q \times q$ diagonal submatrices and $\underline{Y}$ is a $p \times p$ diagonal array of $q \times q$ submatrices. The entries along the diagonal of $\underline{Y}$ are all identical and equal to $\underline{M}_{q,\alpha^p}$. The $q \times q$ diagonal submatrix on the ith row and jth column of the $p \times p$ array comprising $\underline{X}$ $(i,j = 0,1,\ldots,p-1)$ is $\mathrm{diag}[\alpha^{qij}, \alpha^{(qi+1)j}, \ldots, \alpha^{(qi+q-1)j}]$ as in Figure 1.

Since p is the smallest divisor of n, the elements $\alpha,\alpha^2,\alpha^3,\ldots,\alpha^{p-1}$ are all primitive nth roots of unity. It follows that $(p-1)$ multiplications by each of $1,\alpha,\alpha^2,\ldots,\alpha^{n-1}$ and $n(p-1)$ additions are required to compute $\underline{A}\ \underline{X}$. The total hardware for this part is $(p-1)m^2 2^{m-1}$ XOR gates and the delay is $\lceil \log_2 m \rceil + \lceil \log_2 p \rceil$ gate delays. Of course, we still have to multiply by $\underline{Y}$, but this is equivalent to computing p transforms of length q. If q is not a prime, let p' be the smallest nontrivial divisor of q and let $q = p'q'$. Proceeding with the factorisation of $\underline{M}_{q,\alpha^p}$ exactly as before, we get $\underline{M}^*_{q,\alpha^p} = \underline{X}'\underline{Y}'$ etc. To multiply by $\underline{X}'$ requires $(p'-1)$ multiplications by each of $1, \alpha^p, \alpha^{2p}, \ldots, \alpha^{p(q-1)}$ and $q(p'-1)$ additions. Now $\alpha^p = \gamma$ is a qth root of unity. The total number of bit operations required to multiply q elements of $GF(2^m)$ by $1,\gamma,\gamma^2,\ldots,\gamma^{q-1}$ respectively and sum the results is given in column 5 of Table 1 (the heading on the column will be explained later). Hence, we can determine the hardware required to multiply by $\underline{X}'$ (remember that p transforms of length q have to be computed), and proceed to determine the

$$\underline{X} = \begin{bmatrix} I & \begin{bmatrix} 1 & & & & \\ & \alpha & & & O \\ & & \alpha^{2} & & \\ & O & & \ddots & \\ & & & & \alpha^{q-1} \end{bmatrix} & \begin{bmatrix} 1 & & & & \\ & \alpha^{2} & & & O \\ & & \alpha^{4} & & \\ & O & & \ddots & \\ & & & & \alpha^{2(q-1)} \end{bmatrix} & \cdots & \begin{bmatrix} 1 & & & & \\ & \alpha^{p-1} & & & O \\ & & \alpha^{2(p-1)} & & \\ & O & & \ddots & \\ & & & & \alpha^{(p-1)(q-1)} \end{bmatrix} \\ I & \begin{bmatrix} \alpha^{q} & & & O \\ & \alpha^{q+1} & & \\ O & & \ddots & \\ & & & \alpha^{2q-1} \end{bmatrix} & \begin{bmatrix} \alpha^{2q} & & & O \\ & \alpha^{2q+2} & & \\ O & & \ddots & \\ & & & \alpha^{4q-2} \end{bmatrix} & \cdots & \begin{bmatrix} \alpha^{(p-1)q} & & & O \\ & \alpha^{(p-1)q+p-1} & & \\ O & & \ddots & \\ & & & \alpha^{(p-1)(2q-1)} \end{bmatrix} \\ \vdots & \vdots & \vdots & & \vdots \\ I & \begin{bmatrix} \alpha^{q(p-1)} & & & O \\ & \alpha^{q(p-1)+1} & & \\ O & & \ddots & \\ & & & \alpha^{pq-1} \end{bmatrix} & \begin{bmatrix} \alpha^{2q(p-1)} & & & O \\ & \alpha^{2q(p-1)+2} & & \\ O & & \ddots & \\ & & & \alpha^{2(pq-1)} \end{bmatrix} & \cdots & \begin{bmatrix} \alpha^{(p-1)(pq-q)} & & & O \\ & \alpha^{(p-1)(pq-q+1)} & & \\ O & & \ddots & \\ & & & \alpha^{(p-1)(pq-1)} \end{bmatrix} \end{bmatrix}$$

Figure 1. The FFT matrix $\underline{X}$. I is a q $\times$ q identity matrix. All submatrices of X are diagonal submatrices.

the total number of bit operations required to compute the Fourier Transform using the FFT. If $n = n_1 n_2 n_3 \ldots n_s$, the overall delay is at most $s\lceil \log_2 m\rceil + \lceil \log_2 n_1\rceil + \ldots + \lceil \log_2 n_s\rceil$ gate delays. It is also easy to show that a total of $n(n_1+n_2+\ldots+n_s-s)$ additions and $n(n_1+n_2+\ldots+n_s-s-1)+1$ multiplications (by quantities other than 1) are required. However, in order to count bit operations, we have to consider the orders of the elements involved.

Let us now consider the S-FFT algorithm. First, let $n = 2^m-1$ be a prime. We note that $(n-1)$ additions, requiring a total of $m(n-1)$ XOR gates are needed to compute B_0. Since $1, \alpha^j, \alpha^{2j}, \ldots, \alpha^{(n-1)j}$ are the n distinct nonzero m-tuples, we see that computing

$$B_j^{(k)} = \sum_{i=0}^{n-1} A_{i,k}(\alpha^j)^i$$

requires a total of $m(2^{m-1}-1)$ XOR gates and a delay of $\lceil \log_2 2^{m-1}\rceil = (m-1)$ gate delays. Such computations are required for m values of k and $I(n) = (n-1)/m$ values of j. Now, the squaring operations in (6) can also be implemented by linear transformations (e.g., [7], p. 50) as can the multiplications in (5).

The numbers of ones in matrices corresponding to squaring can be made quite small if the minimal polynomial of α is chosen carefully. The numbers of gates and the delays in a squaring circuit are given in Table 2. For example, if $m = 5$, $n = 31$, then one can successively compute $B_2^{(k)}$, $B_4^{(k)}$, $B_8^{(k)}$ and $B_{16}^{(k)}$ from $B_1^{(k)}$ using a cascade of four squaring circuits, giving a total of 12 gates and a delay of 4 gate delays. Alternatively, since squaring is a linear operation, so is taking the 4th, 8th and 16th power of an element. Hence, $B_1^{(k)}$ can be applied to four different circuits

m	n	Minimal Polynomial of α	Squaring Circuit		Multiplier Circuits			
					Horner's rule		Simultaneous Mult.	
			# of gates	Delay	# of gates	Delay	# of gates	Max. delay
3	7	x^3+x+1	1	1	2	2	3	1
4	15	x^4+x+1	2	1	3	3	6	1
5	31	x^5+x^2+1	3	1	4	4	11	2
6	63	x^6+x+1	3	1	5	5	15	1
7	127	x^7+x+1	3	1	6	6	21	1
8	255	$x^8+x^4+x^3+x^2+1$	20	2	21	7	86	2
9	511	x^9+x^4+1	6	1	8	8	42	2
10	1023	$x^{10}+x^3+1$	6	2	9	9	48	2
11	2047	$x^{11}+x^2+1$	6	1	10	10	56	2

Table 2. Numbers of gates and delays required to implement (a) the squaring operation (b) the (m-1) multiplications in Horner's rule and (c) simultaneous multiplications by $1,\alpha,\ldots,\alpha^{m-1}$. Gates and delays in adder circuits are not included.

to produce $B_2^{(k)}$, $B_4^{(k)}$, $B_8^{(k)}$ and $B_{16}^{(k)}$ simultaneously. It can be shown that this requires 23 gates but the delay is only 2 gate delays.

A similar situation exists in computing the B_j's from the $B_j^{(k)}$'s. Using Horner's rule, we can compute B_j from the $B_j^{(k)}$'s using (m-1) multiplications by α and (m-1) additions. The total number of XOR gates and delays required to implement all (m-1) multiplications by α are given in Table 2. We also need (m-1) adders, which require m(m-1) XOR gates and delay the signals by a further (m-1) gate delay. Alternatively, one can multiply each $B_j^{(k)}$ by α^k $(k = 0,1,2,\ldots,m-1)$ simultaneously and add the products. The total number of gates and the maximum delay is listed in Table 2. As before, (m-1) adders are required but the increase in delay is only $\lceil \log_2 m \rceil$ gate delays. While discussing the direct method, we saw that we could decrease the delay without increasing the hardware by changing to simultaneous multiplications. This is not true here. Except for the case m = 8, exactly one gate is required to multiply by α giving a total of (m-1) gates to implement Horner's rule, while the (m-1) simultaneous multiplications require approximately $\frac{1}{2}m(m-1)$ gates. Even so, these latter multiplications require very little hardware since the <u>average</u> multiplication by a nth root requires $\frac{1}{2}m^2-m$ bit operations, compared to a <u>total</u> of $\frac{1}{2}m(m-1)$ bit operations for (m-1) multiplications. The numbers of bit operations required to implement the S-FFT algorithm can be determined from the above.

When n is not a prime, we have, as before, that the hardware required to compute $B_j^{(k)}$ depends on the order of α^j. This is given in column 6 of Table 1. Using these numbers and Table 2, it is straightforward to determine the total numbers of bit operations required for the S-FFT algorithm.

A technique that I call folding can be used to reduce the number of bit operations at the expense of increased delay for the S-FFT algorithm and for direct computation, whenever n is not a prime. Suppose $\alpha^j = \gamma$ is an element of order d where $d|n$. Then

$$B_j = \sum_{i=0}^{n-1} A_i \gamma^i = \sum_{i=0}^{d-1} \left(\sum_{j=0}^{\frac{n}{d}-1} A_{dj+i} \right) \gamma^i$$

If we precompute the d inner sums, then d multiplications (by $1, \gamma, \gamma^2, \ldots, \gamma^{d-1}$) and d-1 additions suffice to compute B_j. I call the process of computing the inner sums a folding of A(x) to length d and denote the polynomial

$$\sum_{i=0}^{d-1} \left(\sum_{j=0}^{\frac{n}{d}-1} A_{dj+i} \right) x^i \text{ by } A(x|d).$$

Thus, we save (n-d) multiplications and (n-d) additions for $\phi(d)$ different values of j but need to first fold A(x) which requires n-d extra additions. Actually, the savings are even more as we illustrate by an example. Suppose n = 63. Then, we need to find A(x|21) A(x|9), A(x|7) and A(x|3). To compute the first two requires 42 and 54 additions respectively. To find A(x|7), we can either fold A(x) to length 7 using 56 additions, or fold A(x|21) to length 7 using only 14 additions. Obviously, the latter is preferable. Similarly, we can fold A(x|9) to get A(x|3) using only 6 additions. Finally, we can fold A(x|3) to length 1 using 2 additions and this is exactly the same as computing B_0! Consequently, only 42 + 14 <u>extra</u> additions are required for folding operations since the 54 + 6 + 2 additions to fold A(x) to lengths 9, 3 and 1 are required for computing B_0 anyway. However, we save some multiplications and additions in computing the B_j's and the net result is a saving of 1284 multiplications and 1228 additions. This is approximately one-third of the total number of multiplications and additions required for direct computation when $n = 63$. The total hardware required to compute B_j from $A(x|d)$ is given in column 5 of Table 1. The savings due to folding occur in the S-FFT algorithm also. Thus, after folding $A(x)$ to all the necessary lengths, we can compute $B_j^{(k)}$ from $A^{(k)}(x|d)$ which is available directly from $A(x|d)$. The number of bit operations required for this are given in column 7 of Table 1. From this, we can determine the total numbers of bit operations required for the S-FFT algorithm.

In Table 3, we give the total numbers of bit operations required for computing the Fourier Transform by the various algorithms. When n is a prime, there is no FFT algorithm; when n is not a prime, we give the results both with and without folding. We notice, that as might be expected, the FFT is superior to the S-FFT for larger values of n, while at shorter lengths, the S-FFT with folding is superior to the FFT. The S-FFT algorithm is superior to direct computation by a factor of $(m-1)$ approximately. Folding reduces the bit complexity by a factor of $\phi(n)/n$ approximately, i.e., the number of bit operations to evaluate $B_j^{(k)}$ after folding is quite small compared to the number required when folding is not used. Finally, the case $n = 255$ is exceptional in that the squaring circuits and multiplier circuits require unusually large amounts of hardware and hence the S-FFT requires more hardware than the FFT.

n	Algorithm	Number of bit Operations
7	Direct	216
	S-FFT	132
15	Direct	1,788
	Direct with folding	1,284
	FFT	760
	S-FFT	778
	S-FFT with folding	642
31	Direct	12,000
	S-FFT	3,480
63	Direct	71,412
	Direct with folding	48,300
	FFT	11,736
	S-FFT	16,822
	S-FFT with folding	11,494
127	Direct	395,136
	S-FFT	64,764
255	Direct	2,069,788
	Direct with folding	1,292,460
	FFT	178,336
	S-FFT	328,518
	S-FFT with folding	219,766
511	Direct	10,565,952
	Direct with folding	9,177,126
	FFT	1,620,288
	S-FFT	1,259,292
	S-FFT with folding	1,088,706
1023	Direct	52,290,464
	Direct with folding	36,222,784
	FFT	2,103,520
	S-FFT	5,548,398
	S-FFT with folding	3,798,638
2047	Direct	253,453,376
	Direct with folding	240,403,042
	FFT	13,606,912
	S-FFT	23,381,336
	S-FFT with folding	22,241,274

Table 3. Numbers of bit operations required to compute the Fourier Transform by various algorithms.

IV. Implementations and Applications

The numbers of bit operations required to compute the Fourier Transform using the various algorithms was discussed at some length in the previous section. The nm-input, nm-output combinational network discussed there is not necessarily a practical implementation. Two more realistic methods are a m-input, nm-output sequential network and a nm-input, m-output sequential network. In the former, the input to the circuit is the sequence A_{n-1}, $A_{n-2},\ldots,A_0$. When the last symbol is received, the nm outputs are the B_j's. In the latter, the input is $A_{n-1},A_{n-2},\ldots,A_0$ which is stored in the network. The network then successively generates $B_0,B_1,\ldots,B_{n-1}$. In the coding literature, these are known as syndrome generating circuits and Chien search circuits respectively [7, Chapter 5], [9]. Unfortunately, the S-FFT algorithm is not well-suited to either of these methods. It requires approximately the same number of flip flops but considerably more XOR gates than the direct method. I am not aware of any implementation of the FFT algorithm along these lines. It appears to be even more difficult to implement than the S-FFT.

An alternative implementation is in software with multiplication being done by means of tables of logarithms and antilogarithms in $GF(2^m)$. As discussed earlier, the direct method could be implemented using $(n-1)^2$ multiplications and $n(n-1)$ additions while the FFT would require $n(n_1+n_2+\ldots+n_s-s-1)+1$ multiplications and $n(n_1+n_2+\ldots+n_s-s)$ additions. For the S-FFT, one can trade-off time for memory space. Thus, one has to compute $B_j^{(k)}$ for m values of k and I(n) values of j, say $j_1,j_2\ldots,j_{I(n)}$. One can store a $(n-1) \times I(n)$ array whose i^{th} row is $\alpha^{ij_1},\alpha^{ij_2},\ldots,\alpha^{ij_{I(n)}}$ and a $m \times I(n)$ array containing the partial sums for $B_j^{(k)}$. Then, after testing $A_{i,k}$, one either adds (or does not add) the i^{th} row of the first array into the k^{th} row of the second. Furthermore, if one stores two tables (with 2^m entries each) whose γ^{th} entries are respectively γ^2 and $\gamma\beta$, one can dispense with the log and antilog tables entirely. As in the hardware implementation, the squarings and multiplications by β in (5) are less complex than multiplications in general, since the former require only one table lookup while the latter require two. If one wishes to avoid storing a $(n-1) \times I(n)$ array, then one can compute $\alpha^{ij_1},\ldots,\alpha^{ij_{I(n)}}$ at the i^{th} step. In this case, the S-FFT requires approximately n^2/m extra multiplications. As a result, it cannot always compete with the FFT when n is not a prime; when n is a prime, the S-FFT is still better than the direct method.

In recent years, considerable attention has been paid to Fast Fourier Transforms over finite mathematical structures because no round-off or truncation errors can occur in such transforms. However, the major application envisioned for the S-FFT algorithm (and, indeed, the initial impetus for considering it) is in the encoding and decoding of BCH and Reed-Solomon Codes [7]. Mandelbaum [10] has proposed a technique for implementing Reed-Solomon codes which requires both the encoder and the decoder to compute a Fourier Transform. It has been shown [11], [12] that if an FFT is used for these transforms, then Mandelbaum's technique is superior over the usual implementation for a wide range of rates and block lengths. Use of the S-FFT increases this range and also allows reasonable implementations when the block length is a prime.

For the usual implementations, one must compute syndromes i.e., evaluate a polynomial of degree n-1 at α^j, $j = 1,2,\dots,2t$ and locate errors by the Chien search, i.e., find the Fourier Transform of a polynomial of degree t. For binary BCH codes, the polynomial of degree n-1 is a polynomial over GF(2) and the S-FFT has no application. Equation (6) is used to compute some of the syndrome values but this result is well known in the coding literature. However, fast techniques can be used for syndrome computation of Reed-Solomon codes and Chien search for both BCH and Reed-Solomon codes. Here, direct computation requires approximately 2nt multiplications and additions, and nt multiplications and additions respectively. The reduction in FFT complexity is much smaller. If one thinks of the FFT as multiplication of a row vector by a succession of sparse matrices, then (for the Chien search), a vector with (t+1) nonzero entries becomes a vector with $n_1(t+1)$ nonzero entries after one matrix multiplication and a vector with $n_1n_2(t+1)$ nonzero entries after the second matrix multiplication etc. so that the advantages of beginning with a polynomial of small degree are rapidly lost. On the other hand, for the S-FFT Chien search, one requires $mtI(n) + nm + t$ additions and 2nm multiplications approximately. Similar savings are available in syndrome computation as well. In short, for these specialized cases, direct computation and the S-FFT have reduced complexity in similar ratios while the FFT has not. This is an added advantage for the S-FFT algorithm.

Acknowledgments

The author would like to thank F. P. Preparata for helpful discussions. Work leading to a preliminary version of this algorithm was also supported in part by the Rome Air Development Center under Contract F30602-72-C-0031 and is reported in [12].

References

1. J. M. Pollard, "The Fast Fourier Transform in a Finite Field," Mathematics of Computation, Vol. 25, No. 114, pp. 365-374, April 1971.

2. A. V. Oppenheim and R. W. Schafer, Digital Signal Processing, Prentice Hall, 1975.

3. C. M. Rader, "Discrete Fourier Transforms when the Number of Data Samples is Prime," Proc. IEEE (Letters), Vol. 56, No. 6, pp. 1107-1108, June 1968.

4. L. L. Bluestein, "A Linear Filtering Approach to the Computation of Discrete Fourier Transform," IEEE Trans. Audio & Electroaccoustics, Vol. AU-18, pp. 451-455, Dec. 1970.

5. L. R. Rabiner, R. W. Schafer and C. M. Rader, "The Chirp z-transform Algorithm," IEEE Trans. Audio & Electroaccoustics, Vol. AU-17, pp. 86-92, June 1969.

6. A. A. Albert, Fundamental Concepts of Higher Algebra, University of Chicago Press, 1956.

7. E. R. Berlekamp, Algebraic Coding Theory, McGraw-Hill, 1968.

8. S. W. Golomb, Shift Register Sequences, Holden-Day, 1967.

9. R. T. Chien, "Cyclic Decoding Procedures for Bose-Chaudhari-Hocquenghem Codes," IEEE Transactions on Information Theory, Vol. IT-10, pp. 357-363, October 1964.

10. D. Mandelbaum, "On Decoding of Reed-Solomon Codes," IEEE Transactions on Information Theory, Vol. IT-17, pp. 707-712, November 1971.

11. R. H. Paschburg, "Software Implementation of Error-Correcting Codes," Coordinated Science Laboratory Technical Report R-659, University of Illinois, Urbana, August 1974.

12. R. T. Chien, R. B. Brown, C. L. Chen and D. V. Sarwate, "Hardware and Software Error Correction Coding," Rome Air Development Center, Technical Report RADC-TR-75-217, Griffiss AFB, New York, August 1975.